Figures

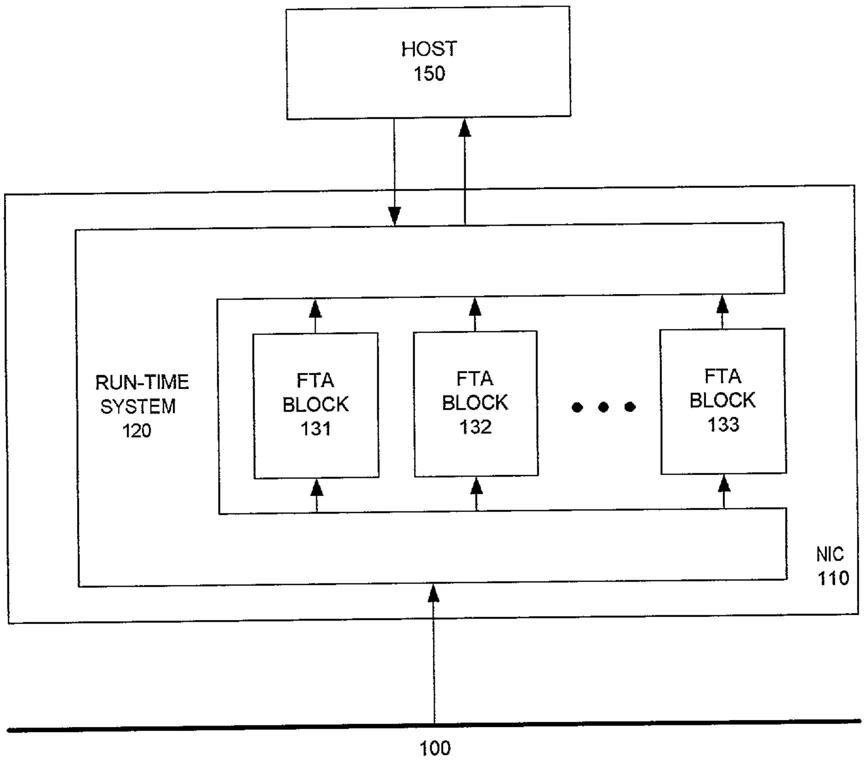

HOST
150
RUN-TIME
SYSTEM
120
FTA
BLOCK
131
FTA
BLOCK
132
FTA
BLOCK
133
NIC
110
100

FIG. 1

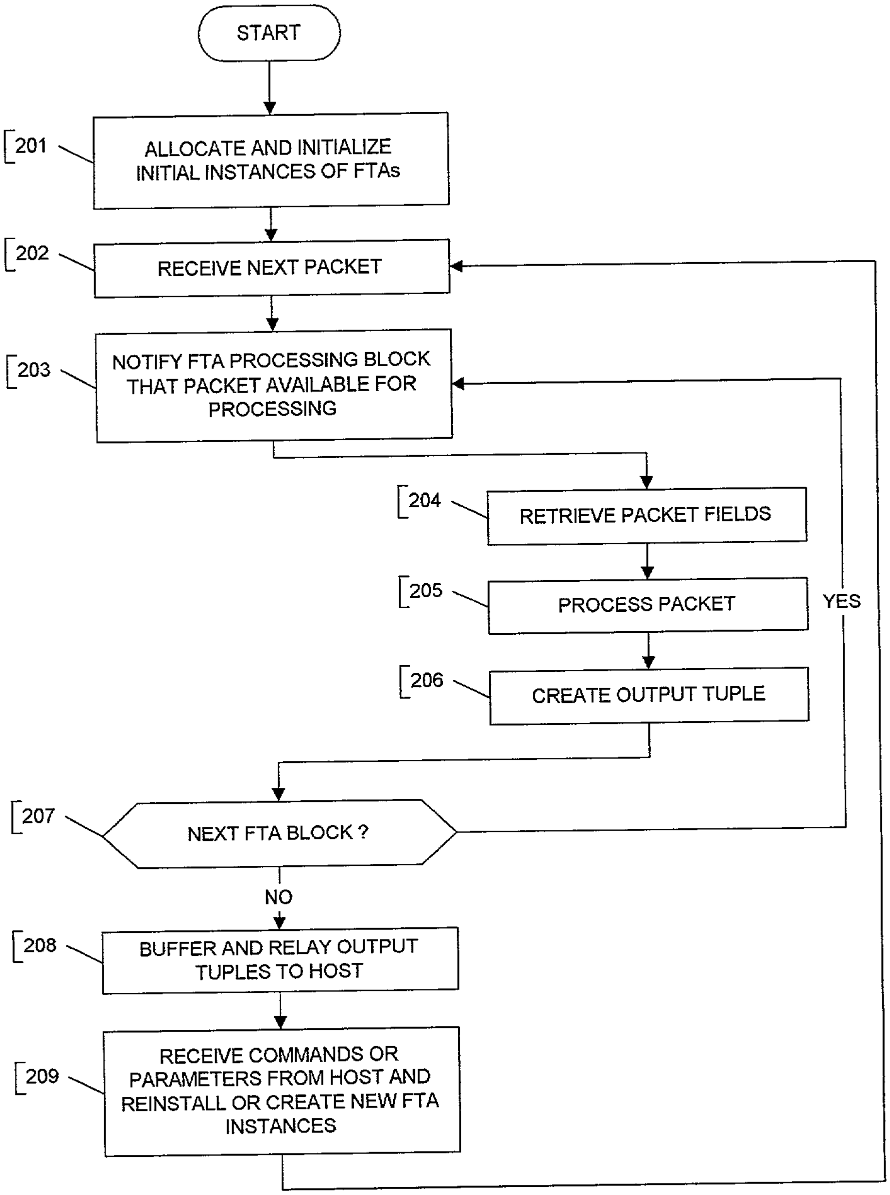
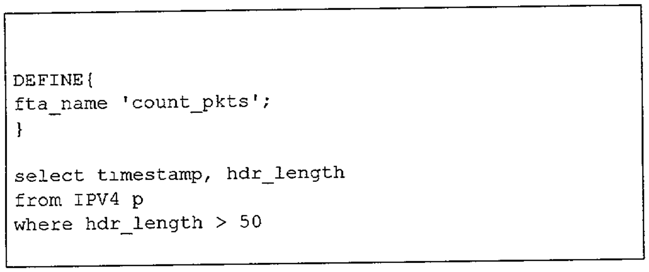
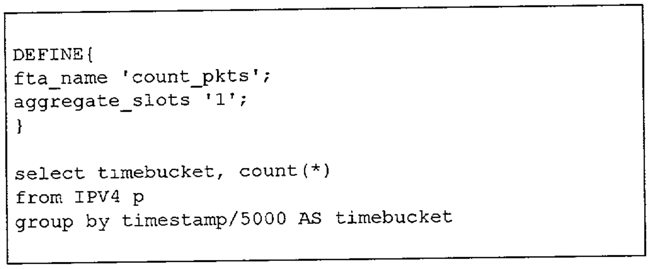

START
201
ALLOCATE AND INITIALIZE INITIAL INSTANCES OF FTAs
202
RECEIVE NEXT PACKET
203
NOTIFY FTA PROCESSING BLOCK THAT PACKET AVAILABLE FOR PROCESSING
204
RETRIEVE PACKET FIELDS
205
PROCESS PACKET
YES
206
CREATE OUTPUT TUPLE
207
NEXT FTA BLOCK ?
NO
208
BUFFER AND RELAY OUTPUT TUPLES TO HOST
209
RECEIVE COMMANDS OR PARAMETERS FROM HOST AND REINSTALL OR CREATE NEW FTA INSTANCES

FIG. 2



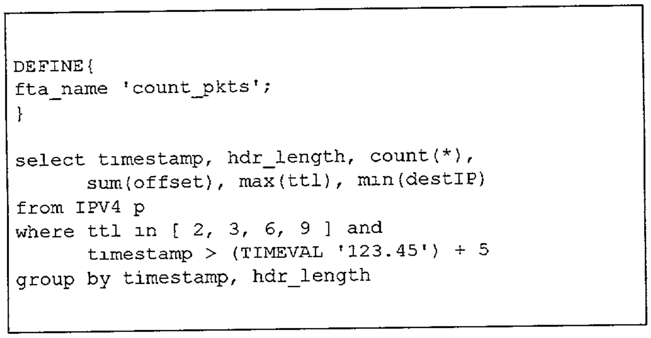
```
DEFINE{
fta_name 'count_pkts';
}

select timestamp, hdr_length
from IPV4 p
where hdr_length > 50
```

FIG. 3



```
DEFINE{
fta_name 'count_pkts';
aggregate_slots '1';
}

select timebucket, count(*)
from IPV4 p
group by timestamp/5000 AS timebucket
```

FIG. 6



```
DEFINE{
fta_name 'count_pkts';
}

select timestamp, hdr_length, count(*),
      sum(offset), max(ttl), min(destIP)
from IPV4 p
where ttl in [ 2, 3, 6, 9 ] and
      timestamp > (TIMEVAL '123.45') + 5
group by timestamp, hdr_length

```

FIG. 9

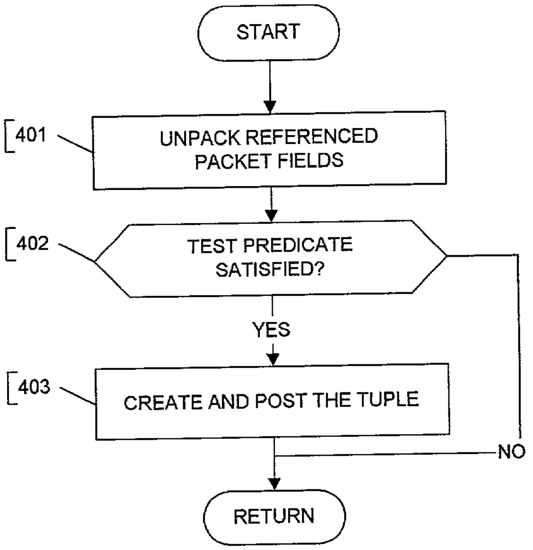
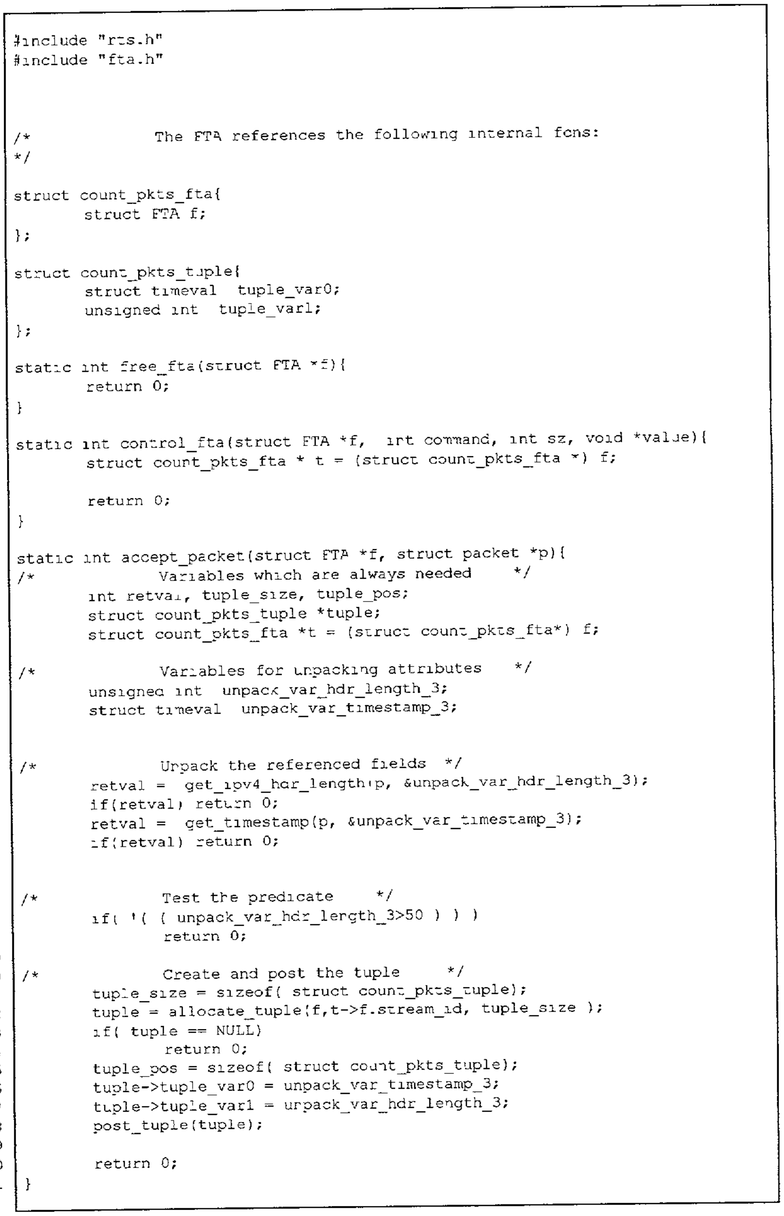
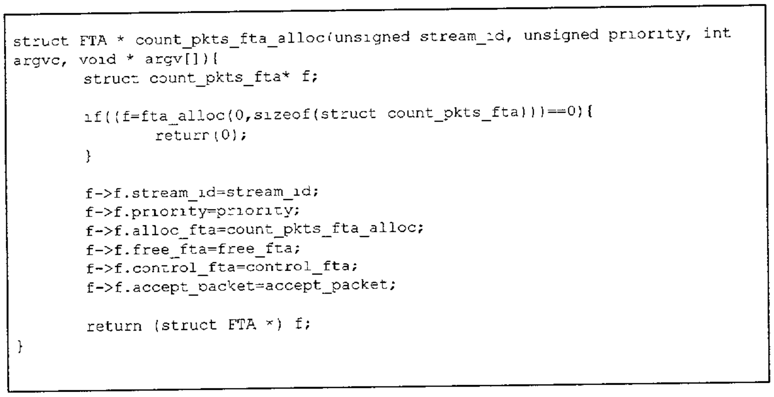

START
401
UNPACK REFERENCED PACKET FIELDS
402
TEST PREDICATE SATISFIED?
YES
403
CREATE AND POST THE TUPLE
NO
RETURN

FIG. 4



```
#include "rts.h"
#include "fta.h"



/*              The FTA references the following internal fcns:
*/

struct count_pkts_fta{
        struct FTA f;
};

struct count_pkts_tuple{
        struct timeval  tuple_var0;
        unsigned int  tuple_var1;
};

static int free_fta(struct FTA *f){
        return 0;
}

static int control_fta(struct FTA *f,  int command, int sz, void *value){
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;

        return 0;
}

static int accept_packet(struct FTA *f, struct packet *p){
/*              Variables which are always needed     */
        int retval, tuple_size, tuple_pos;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta *t = (struct count_pkts_fta*) f;

/*              Variables for unpacking attributes    */
        unsigned int  unpack_var_hdr_length_3;
        struct timeval  unpack_var_timestamp_3;


/*              Unpack the referenced fields  */
        retval =  get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
        if(retval) return 0;
        retval =  get_timestamp(p, &unpack_var_timestamp_3);
        if(retval) return 0;


/*              Test the predicate     */
        if( !( ( unpack_var_hdr_length_3>50 ) ) )
                return 0;

/*              Create and post the tuple      */
        tuple_size = sizeof( struct count_pkts_tuple);
        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
        if( tuple == NULL)
                return 0;
        tuple_pos = sizeof( struct count_pkts_tuple);
        tuple->tuple_var0 = unpack_var_timestamp_3;
        tuple->tuple_var1 = unpack_var_hdr_length_3;
        post_tuple(tuple);

        return 0;
}
```

FIG. 5A



```
struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
argvc, void * argv[]){
        struct count_pkts_fta* f;

        if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0){
                return(0);
        }

        f->f.stream_id=stream_id;
        f->f.priority=priority;
        f->f.alloc_fta=count_pkts_fta_alloc;
        f->f.free_fta=free_fta;
        f->f.control_fta=control_fta;
        f->f.accept_packet=accept_packet;

        return (struct FTA *) f;
}
```

FIG. 5B

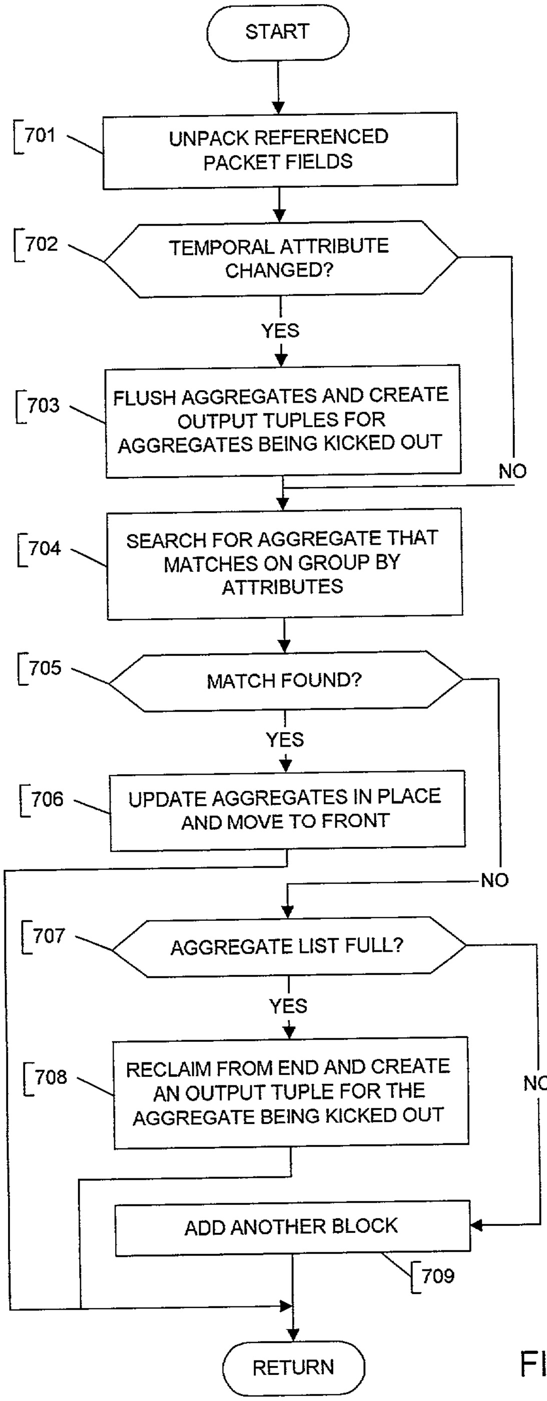
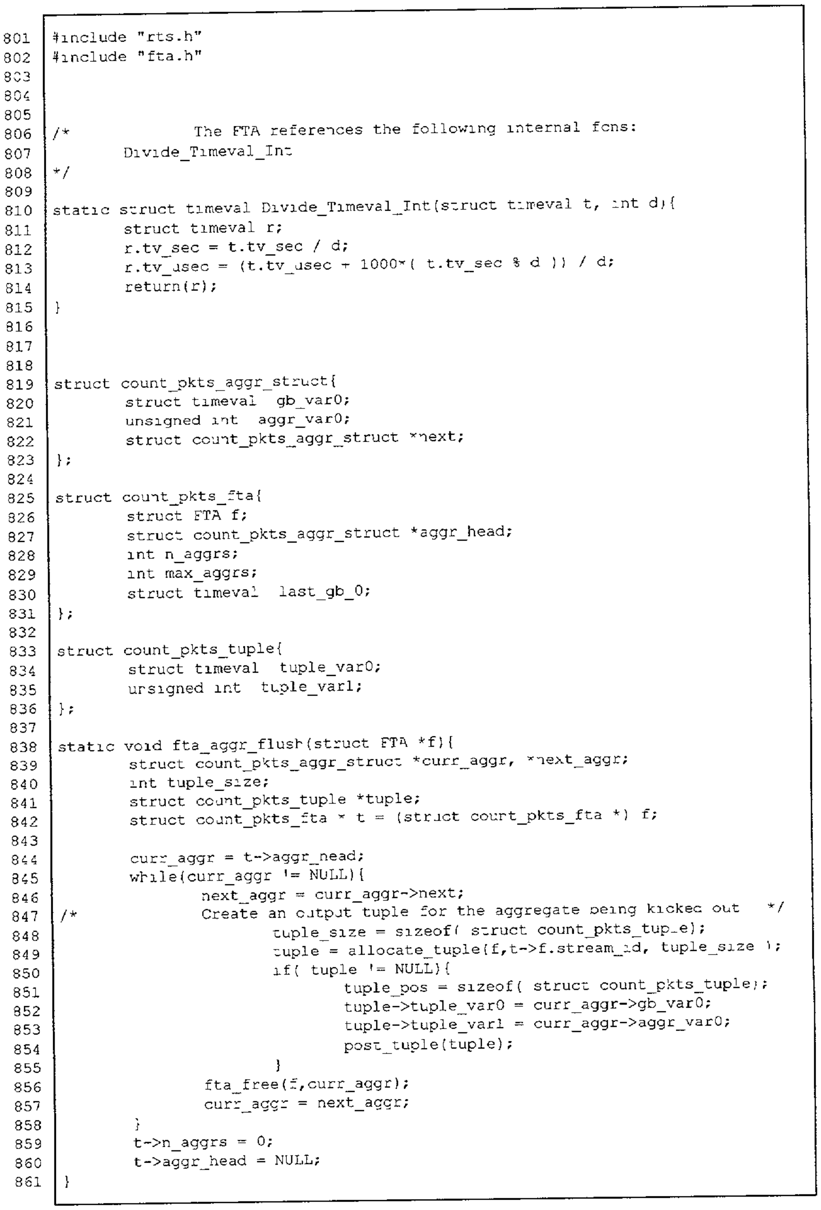

START
701
UNPACK REFERENCED PACKET FIELDS
702
TEMPORAL ATTRIBUTE CHANGED?
YES
NO
703
FLUSH AGGREGATES AND CREATE OUTPUT TUPLES FOR AGGREGATES BEING KICKED OUT
704
SEARCH FOR AGGREGATE THAT MATCHES ON GROUP BY ATTRIBUTES
705
MATCH FOUND?
YES
NO
706
UPDATE AGGREGATES IN PLACE AND MOVE TO FRONT
707
AGGREGATE LIST FULL?
YES
NO
708
RECLAIM FROM END AND CREATE AN OUTPUT TUPLE FOR THE AGGREGATE BEING KICKED OUT
ADD ANOTHER BLOCK
709
RETURN

FIG. 7

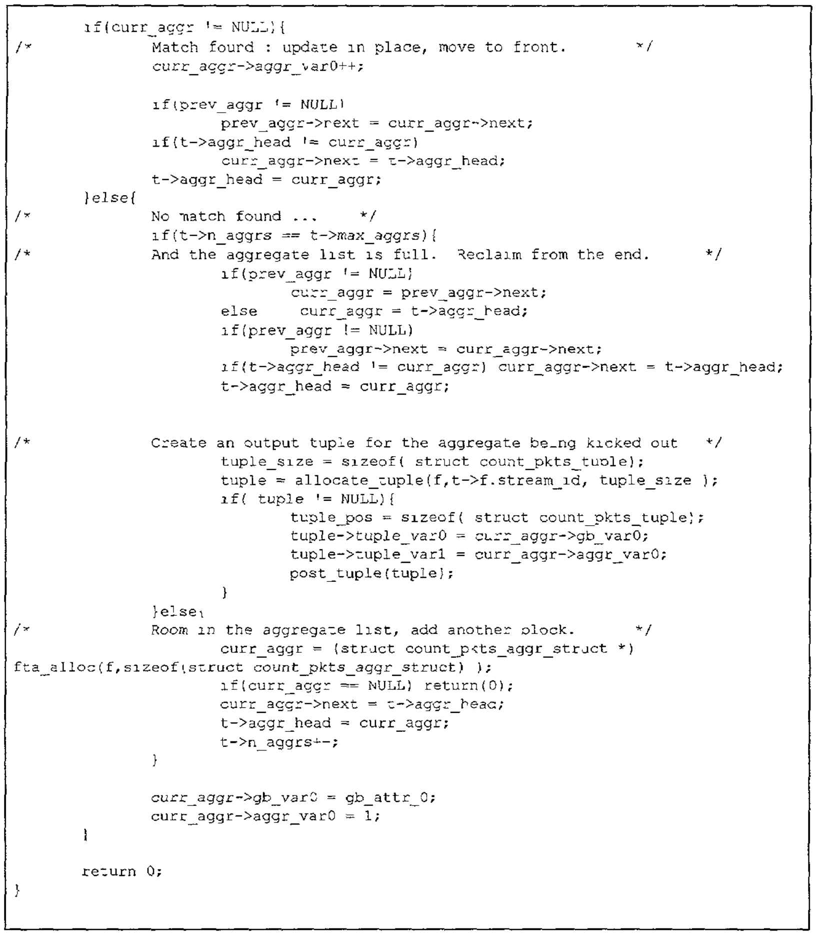


```
#include "rts.h"
#include "fta.h"



/*              The FTA references the following internal fcns:
        Divide_Timeval_Int
*/

static struct timeval Divide_Timeval_Int(struct timeval t, int d){
        struct timeval r;
        r.tv_sec = t.tv_sec / d;
        r.tv_usec = (t.tv_usec + 1000*( t.tv_sec % d )) / d;
        return(r);
}



struct count_pkts_aggr_struct{
        struct timeval  gb_var0;
        unsigned int  aggr_var0;
        struct count_pkts_aggr_struct *next;
};

struct count_pkts_fta{
        struct FTA f;
        struct count_pkts_aggr_struct *aggr_head;
        int n_aggrs;
        int max_aggrs;
        struct timeval  last_gb_0;
};

struct count_pkts_tuple{
        struct timeval  tuple_var0;
        unsigned int  tuple_var1;
};

static void fta_aggr_flush(struct FTA *f){
        struct count_pkts_aggr_struct *curr_aggr, *next_aggr;
        int tuple_size;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;

        curr_aggr = t->aggr_head;
        while(curr_aggr != NULL){
                next_aggr = curr_aggr->next;
/*              Create an output tuple for the aggregate being kicked out   */
                        tuple_size = sizeof( struct count_pkts_tuple);
                        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
                        if( tuple != NULL){
                                tuple_pos = sizeof( struct count_pkts_tuple);
                                tuple->tuple_var0 = curr_aggr->gb_var0;
                                tuple->tuple_var1 = curr_aggr->aggr_var0;
                                post_tuple(tuple);
                        }
                fta_free(f,curr_aggr);
                curr_aggr = next_aggr;
        }
        t->n_aggrs = 0;
        t->aggr_head = NULL;
}
```

FIG. 8A

```
static int free_fta(struct FTA *f){
        fta_aggr_flush();
        return 0;
}

static int control_fta(struct FTA *f,  int command, int sz, void *value){
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;

        if(command == FTA_COMMAND_FLUSH)
                fta_aggr_flush();
        return 0;
}

static int accept_packet(struct FTA *f, struct packet *p){
/*              Variables which are always needed     */
        int retval, tuple_size, tuple_pos;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta *t = (struct count_pkts_fta*) f;

/*              Variables for unpacking attributes    */
        struct timeval  unpack_var_timestamp_3;



/*              Variables for aggregation      */
/*              Group-by attributes    */
        struct timeval  gb_attr_0;

/*              Variables for manipulating the aggregate list        */
        struct count_pkts_aggr_struct *curr_aggr, *prev_aggr;

/*              Unpack the referenced fields  */
        retval =  get_timestamp(p, &unpack_var_timestamp_3);
        if(retval) return 0;


/*               (no predicate to test)        */

/*              Search for an aggregate that matches on the group by attributes     */
        gb_attr_0 = Divide_Timeval_Int(unpack_var_timestamp_3, 5000);

/*              Flush the aggregates if the temporal gb attrs have changed. */
        if( !( (Compare_Timeval(t->last_gb_0, gb_attr_0) == 0) ) )
                fta_aggr_flush();

        curr_aggr = t->aggr_head; prev_aggr = NULL;
        while(curr_aggr != NULL){
                if( (Compare_Timeval(gb_attr_0, curr_aggr->gb_var0) == 0) )
                        break;
                if(curr_aggr->next != NULL)
                        prev_aggr = curr_aggr;
                curr_aggr = curr_aggr->next;
        }

```

FIG. 8B



```
        if(curr_aggr != NULL){
/*              Match fourd : update in place, move to front.          */
                curr_aggr->aggr_var0++;

                if(prev_aggr != NULL)
                        prev_aggr->rext = curr_aggr->next;
                if(t->aggr_head != curr_aggr)
                        curr_aggr->next = t->aggr_head;
                t->aggr_head = curr_aggr;
        }else{
/*              No match found ...     */
                if(t->n_aggrs == t->max_aggrs){
/*              And the aggregate list is full.  Reclaim from the end.       */
                        if(prev_aggr != NULL}
                                curr_aggr = prev_aggr->next;
                        else     curr_aggr = t->aggr_head;
                        if(prev_aggr != NULL)
                                prev_aggr->next = curr_aggr->next;
                        if(t->aggr_head != curr_aggr) curr_aggr->next = t->aggr_head;
                        t->aggr_head = curr_aggr;


/*              Create an output tuple for the aggregate being kicked out   */
                        tuple_size = sizeof( struct count_pkts_tuple);
                        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
                        if( tuple != NULL){
                                tuple_pos = sizeof( struct count_pkts_tuple};
                                tuple->tuple_var0 = curr_aggr->gb_var0;
                                tuple->tuple_var1 = curr_aggr->aggr_var0;
                                post_tuple(tuple};
                        }
                }else{
/*              Room in the aggregate list, add another block.        */
                        curr_aggr = (struct count_pkts_aggr_struct *)
fta_alloc(f,sizeof(struct count_pkts_aggr_struct) );
                        if(curr_aggr == NULL) return(0);
                        curr_aggr->next = t->aggr_head;
                        t->aggr_head = curr_aggr;
                        t->n_aggrs++;
                }

                curr_aggr->gb_var0 = gb_attr_0;
                curr_aggr->aggr_var0 = 1;
        }

        return 0;
}
```

FIG. 8C

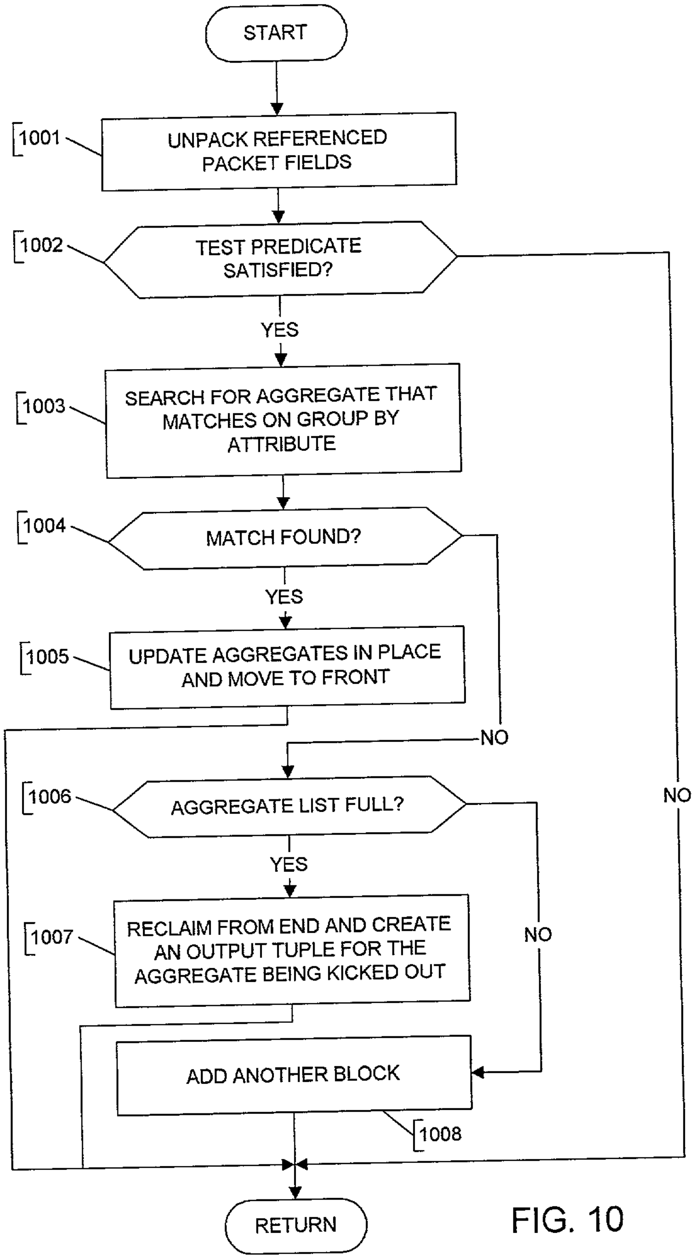
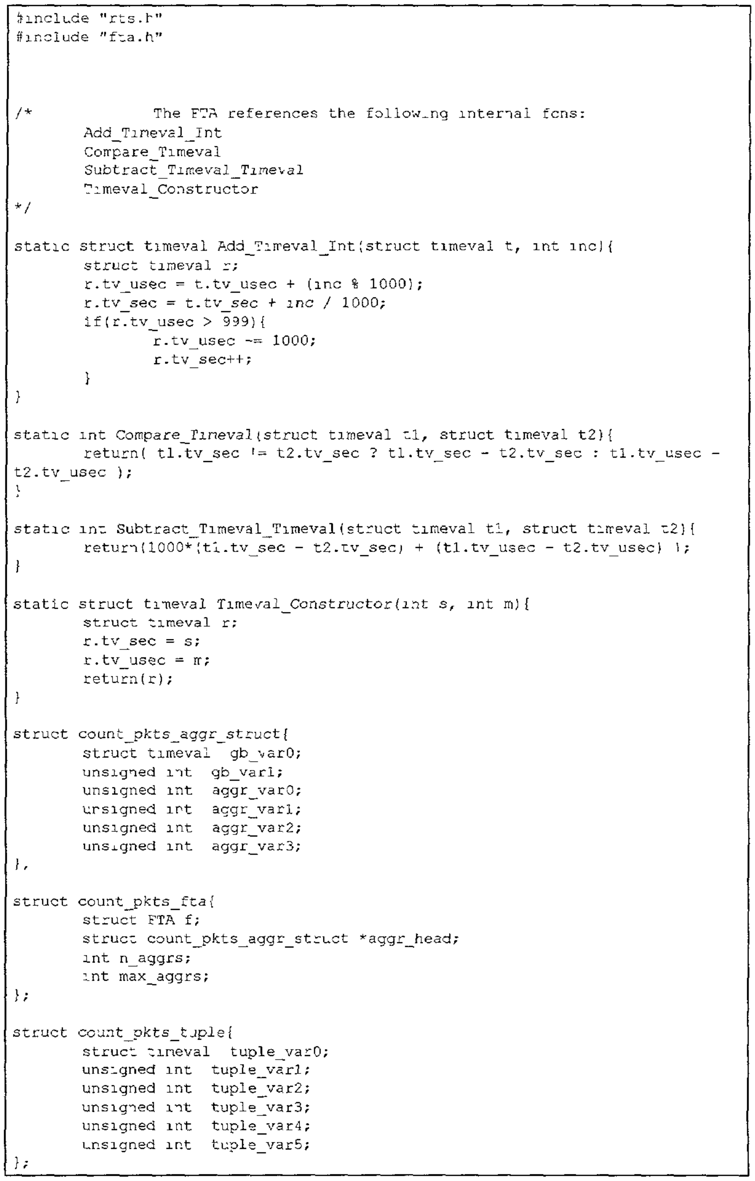
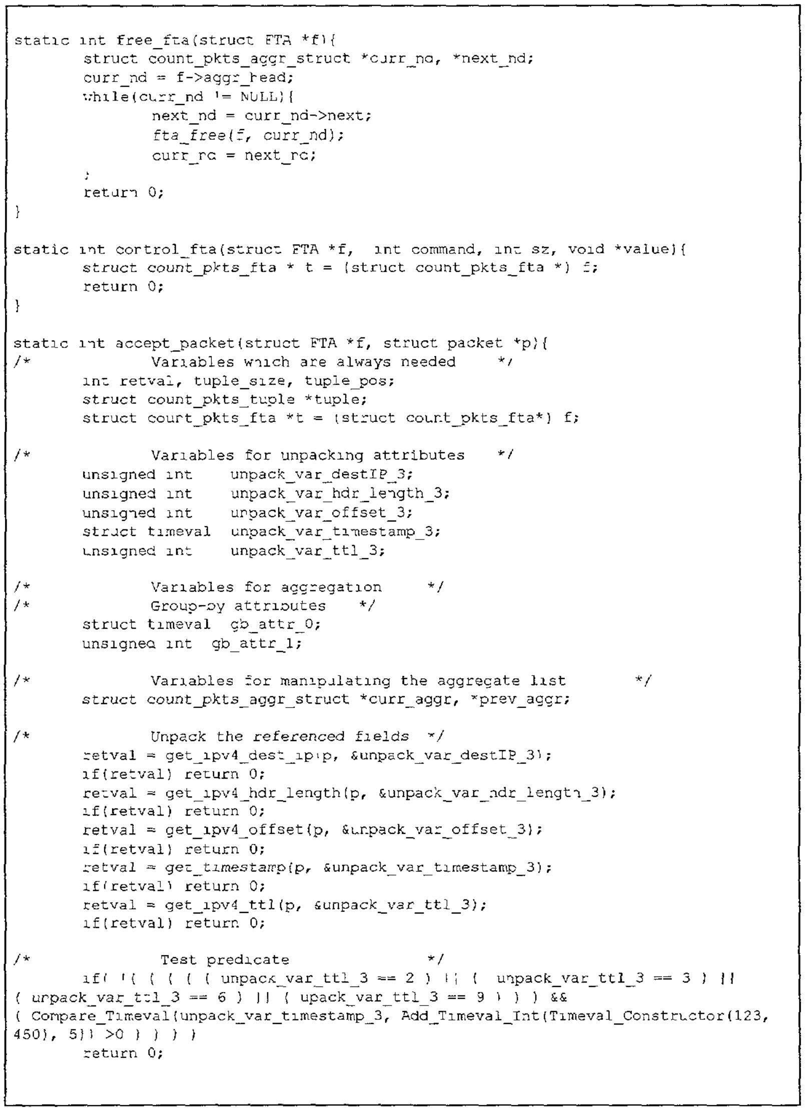

START
1001
UNPACK REFERENCED PACKET FIELDS
1002
TEST PREDICATE SATISFIED?
YES
NO
1003
SEARCH FOR AGGREGATE THAT MATCHES ON GROUP BY ATTRIBUTE
1004
MATCH FOUND?
YES
NO
1005
UPDATE AGGREGATES IN PLACE AND MOVE TO FRONT
1006
AGGREGATE LIST FULL?
YES
NO
1007
RECLAIM FROM END AND CREATE AN OUTPUT TUPLE FOR THE AGGREGATE BEING KICKED OUT
ADD ANOTHER BLOCK
1008
RETURN

FIG. 10



```
#include "rts.h"
#include "fta.h"



/*              The FTA references the following internal fcns:
        Add_Timeval_Int
        Compare_Timeval
        Subtract_Timeval_Timeval
        Timeval_Constructor
*/

static struct timeval Add_Timeval_Int(struct timeval t, int inc){
        struct timeval r;
        r.tv_usec = t.tv_usec + (inc % 1000);
        r.tv_sec = t.tv_sec + inc / 1000;
        if(r.tv_usec > 999){
                r.tv_usec -= 1000;
                r.tv_sec++;
        }
}

static int Compare_Timeval(struct timeval t1, struct timeval t2){
        return( t1.tv_sec != t2.tv_sec ? t1.tv_sec - t2.tv_sec : t1.tv_usec -
t2.tv_usec );
}

static int Subtract_Timeval_Timeval(struct timeval t1, struct timeval t2){
        return(1000*(t1.tv_sec - t2.tv_sec) + (t1.tv_usec - t2.tv_usec) );
}

static struct timeval Timeval_Constructor(int s, int m){
        struct timeval r;
        r.tv_sec = s;
        r.tv_usec = m;
        return(r);
}

struct count_pkts_aggr_struct{
        struct timeval  gb_var0;
        unsigned int  gb_var1;
        unsigned int  aggr_var0;
        unsigned int  aggr_var1;
        unsigned int  aggr_var2;
        unsigned int  aggr_var3;
};

struct count_pkts_fta{
        struct FTA f;
        struct count_pkts_aggr_struct *aggr_head;
        int n_aggrs;
        int max_aggrs;
};

struct count_pkts_tuple{
        struct timeval  tuple_var0;
        unsigned int  tuple_var1;
        unsigned int  tuple_var2;
        unsigned int  tuple_var3;
        unsigned int  tuple_var4;
        unsigned int  tuple_var5;
};
```

FIG. 11A

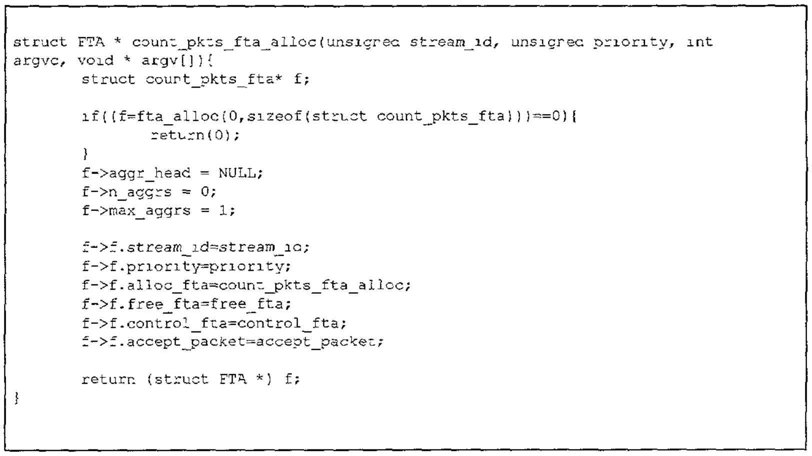


```
static int free_fta(struct FTA *f){
        struct count_pkts_aggr_struct *curr_nd, *next_nd;
        curr_nd = f->aggr_head;
        while(curr_nd != NULL){
                next_nd = curr_nd->next;
                fta_free(f, curr_nd);
                curr_nd = next_nd;
        }
        return 0;
}

static int control_fta(struct FTA *f,  int command, int sz, void *value){
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;
        return 0;
}

static int accept_packet(struct FTA *f, struct packet *p){
/*              Variables which are always needed     */
        int retval, tuple_size, tuple_pos;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta *t = (struct count_pkts_fta*) f;

/*              Variables for unpacking attributes    */
        unsigned int    unpack_var_destIP_3;
        unsigned int    unpack_var_hdr_length_3;
        unsigned int    unpack_var_offset_3;
        struct timeval  unpack_var_timestamp_3;
        unsigned int    unpack_var_ttl_3;

/*              Variables for aggregation      */
/*              Group-by attributes    */
        struct timeval  gb_attr_0;
        unsigned int  gb_attr_1;

/*              Variables for manipulating the aggregate list        */
        struct count_pkts_aggr_struct *curr_aggr, *prev_aggr;

/*              Unpack the referenced fields  */
        retval = get_ipv4_dest_ip(p, &unpack_var_destIP_3);
        if(retval) return 0;
        retval = get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
        if(retval) return 0;
        retval = get_ipv4_offset(p, &unpack_var_offset_3);
        if(retval) return 0;
        retval = get_timestamp(p, &unpack_var_timestamp_3);
        if(retval) return 0;
        retval = get_ipv4_ttl(p, &unpack_var_ttl_3);
        if(retval) return 0;

/*               Test predicate               */
        if( !( ( ( ( ( unpack_var_ttl_3 == 2 ) || (  unpack_var_ttl_3 == 3 ) ||
( unpack_var_ttl_3 == 6 ) || ( upack_var_ttl_3 == 9 ) ) ) &&
( Compare_Timeval(unpack_var_timestamp_3, Add_Timeval_Int(Timeval_Constructor(123,
450), 5)) >0 ) ) ) )
        return 0;
```

FIG. 11B

```
/*              Search for an aggregate that matches on the group by attributes    */
        gb_attr_0 = unpack_var_timestamp_3;
        gb_attr_1 = unpack_var_hdr_length_3;
        curr_aggr = t->aggr_head; prev_aggr = NULL;
        while(curr_aggr != NULL){
                if( (Compare_Timeval(gb_attr_0, curr_aggr->gb_var0) == 0) &&
(gb_attr_1 == curr_aggr->gb_var_1) )
                        break;
                if(curr_aggr->next != NULL)
                        prev_aggr = curr_aggr;
                curr_aggr = curr_aggr->next;
        }

        if(curr_aggr != NULL){
/*              Match found : update in place, move to front.       */
                curr_aggr->aggr_var0++;
                curr_aggr->aggr_var1 += unpack_var_offset_3;
                curr_aggr->aggr_var2 = ( curr_agg->aggr_var2 >= unpack_var_ttl_3 ?
curr_aggr->aggr_var2 : unpack_var_ttl_3 );
                curr_aggr->aggr_var3 = ( curr_agg->aggr_var3 <=
unpack_var_destIP_3 ? curr_aggr->aggr_var3 : unpack_var_destIP_3 );
                if(prev_aggr != NULL)
                        prev_aggr->next = curr_aggr->next;
                if(t->aggr_head != curr_aggr)
                        curr_aggr->next = t->aggr_head;
                t->aggr_head = curr_aggr;
        }else{
/*              No match found ...      */
                if(t->n_aggrs == t->max_aggrs){
/*              And the aggregate list is full.  Reclaim from the end.      */
                        if(prev_aggr != NULL)
                                curr_aggr = prev_aggr->next;
                        else     curr_aggr = t->aggr_head;
                        if(prev_aggr != NULL)
                                prev_aggr->next = curr_aggr->next;
                        if(t->aggr_head != curr_aggr) curr_aggr->next =
t->aggr_head;
                        t->aggr_head = curr_aggr;

/*              Create an output tuple for the aggregate being kicked out   */
                        tuple_size = sizeof( struct count_pkts_tuple);
                        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
                        if( tuple != NULL){
                                tuple_pos = sizeof( struct count_pkts_tuple);
                                tuple->tuple_var0 = curr_aggr->gb_var0;
                                tuple->tuple_var1 = curr_aggr->gb_var1;
                                tuple->tuple_var2 = curr_aggr->aggr_var0;
                                tuple->tuple_var3 = curr_aggr->aggr_var1;
                                tuple->tuple_var4 = curr_aggr->aggr_var2;
                                tuple->tuple_var5 = curr_aggr->aggr_var3;
                                post_tuple(tuple);
                        }
                }else{
/*              Room in the aggregate list, add another block.      */
                        curr_aggr = (struct count_pkts_aggr_struct *)
fta_alloc(f,sizeof(struct count_pkts_aggr_struct) );
                        if(curr_aggr == NULL) return(0);
                        curr_aggr->next = t->aggr_head;
                        t->aggr_head = curr_aggr;
                        t->n_aggrs--;
                }

                curr_aggr->gb_var0 = gb_attr_0;
                curr_aggr->gb_var1 = gb_attr_1;
                curr_aggr->aggr_var0 = 1;
                curr_aggr->aggr_var1 = unpack_var_offset_3;
                curr_aggr->aggr_var2 = unpack_var_ttl_3;
                curr_aggr->aggr_var3 = unpack_var_destIP_3;
        }

        return 0;
}
```

FIG. 11C



```
struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
argvc, void * argv[]){
        struct count_pkts_fta* f;

        if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0){
                return(0);
        }
        f->aggr_head = NULL;
        f->n_aggrs = 0;
        f->max_aggrs = 1;

        f->f.stream_id=stream_id;
        f->f.priority=priority;
        f->f.alloc_fta=count_pkts_fta_alloc;
        f->f.free_fta=free_fta;
        f->f.control_fta=control_fta;
        f->f.accept_packet=accept_packet;

        return (struct FTA *) f;
}
```

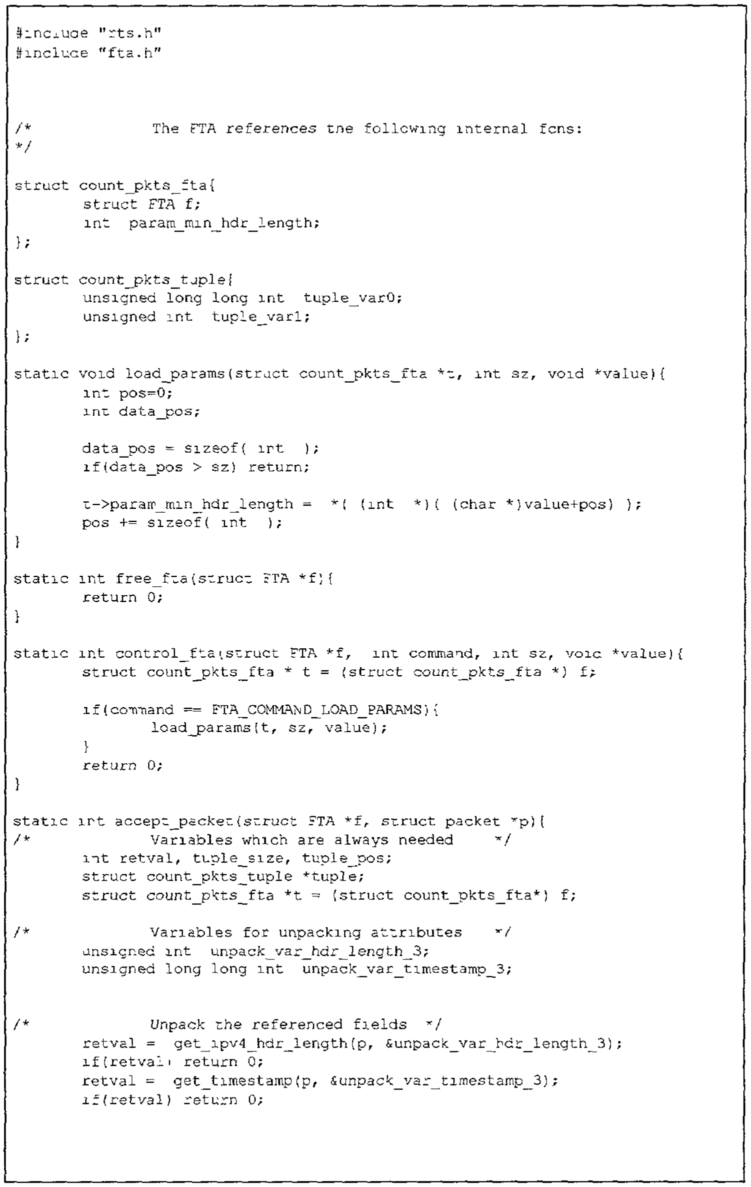
FIG. 11D

```
DEFINE{
fta_name 'test_query';
}

select hdr_length,  max( str_find_substr(IPv4_header, 'bob') ),
            str_find_substr( min(IPv4_header) , 'bob')
from IPV4 p
where precedence > 5 and IPv4_header >
            str_find_substr(IPv4_data, 'host:*\n')
group by hdr_length
```

FIG. 12

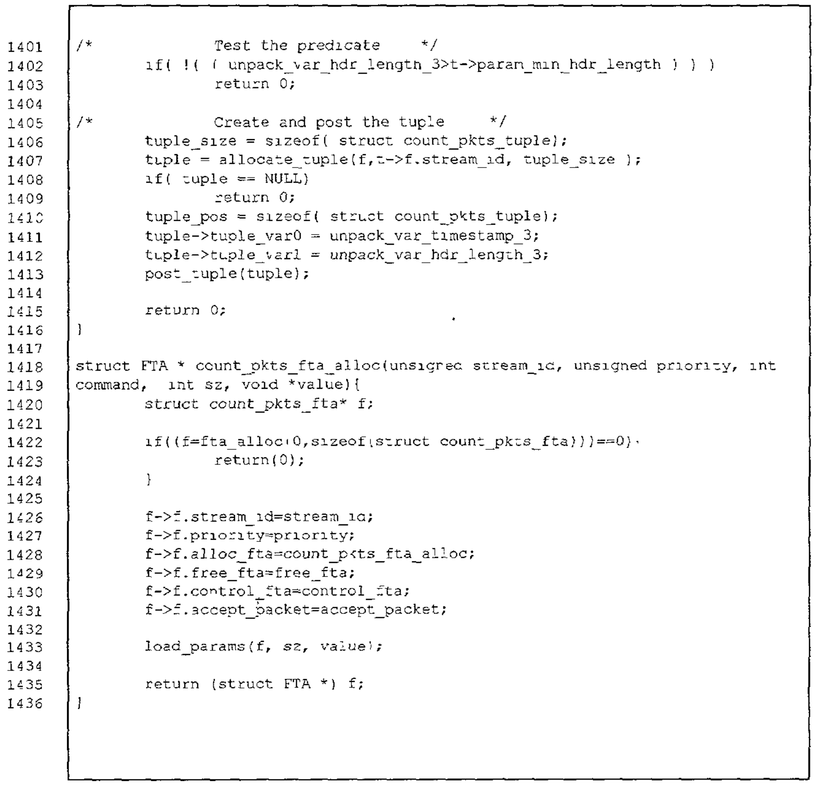
```
DEFINE{
fta_name 'count_pkts';
min_hdr_length 'int';
}

select timestamp, hdr_length
from IPV4 p
where hdr_length > $min_hdr_length
```

FIG. 13



```
#include "rts.h"
#include "fta.h"



/*              The FTA references the following internal fcns:
*/

struct count_pkts_fta{
        struct FTA f;
        int  param_min_hdr_length;
};

struct count_pkts_tuple{
        unsigned long long int  tuple_var0;
        unsigned int  tuple_var1;
};

static void load_params(struct count_pkts_fta *t, int sz, void *value){
        int pos=0;
        int data_pos;

        data_pos = sizeof( int  );
        if(data_pos > sz) return;

        t->param_min_hdr_length =  *( (int  *)( (char *)value+pos) );
        pos += sizeof( int  );
}

static int free_fta(struct FTA *f){
        return 0;
}

static int control_fta(struct FTA *f,  int command, int sz, void *value){
        struct count_pkts_fta * t = (struct count_pkts_fta *) f;

        if(command == FTA_COMMAND_LOAD_PARAMS){
                load_params(t, sz, value);
        }
        return 0;
}

static int accept_packet(struct FTA *f, struct packet *p){
/*              Variables which are always needed     */
        int retval, tuple_size, tuple_pos;
        struct count_pkts_tuple *tuple;
        struct count_pkts_fta *t = (struct count_pkts_fta*) f;

/*              Variables for unpacking attributes    */
        unsigned int  unpack_var_hdr_length_3;
        unsigned long long int  unpack_var_timestamp_3;


/*              Unpack the referenced fields  */
        retval =  get_ipv4_hdr_length(p, &unpack_var_hdr_length_3);
        if(retval) return 0;
        retval =  get_timestamp(p, &unpack_var_timestamp_3);
        if(retval) return 0;
```

FIG. 14A



```
/*              Test the predicate      */
        if( !( ( unpack_var_hdr_length_3>t->param_min_hdr_length ) ) )
                return 0;

/*              Create and post the tuple       */
        tuple_size = sizeof( struct count_pkts_tuple);
        tuple = allocate_tuple(f,t->f.stream_id, tuple_size );
        if( tuple == NULL)
                return 0;
        tuple_pos = sizeof( struct count_pkts_tuple);
        tuple->tuple_var0 = unpack_var_timestamp_3;
        tuple->tuple_var1 = unpack_var_hdr_length_3;
        post_tuple(tuple);

        return 0;
}

struct FTA * count_pkts_fta_alloc(unsigned stream_id, unsigned priority, int
command,  int sz, void *value){
        struct count_pkts_fta* f;

        if((f=fta_alloc(0,sizeof(struct count_pkts_fta)))==0){
                return(0);
        }

        f->f.stream_id=stream_id;
        f->f.priority=priority;
        f->f.alloc_fta=count_pkts_fta_alloc;
        f->f.free_fta=free_fta;
        f->f.control_fta=control_fta;
        f->f.accept_packet=accept_packet;

        load_params(f, sz, value);

        return (struct FTA *) f;
}
```

FIG. 14B